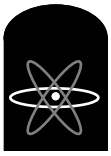
UCRL-ID-113438

Workshop on Developing Safe Software: Final Report

J. Dennis Lawrence

November 30, 1992



FESSP

Fission Energy and Systems Safety Program

Lawrence Livermore National Laboratory

Disclaimer

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Workshop on Developing Safe Software: Final Report

J. Dennis Lawrence

November 30, 1992

Contents

Executive Summary 1
1. Wednesday Morning Prepared Talks 3
1.1. Talk by John Gallagher 3
1.2. Talk by Nancy Leveson 3
1.3. Talk by Bev Littlewood 5
1.4. Talk by Ricky Butler 8
1.5. Talk by John Rushby 9
2. Wednesday Afternoon Discussion 11
2.1. Management Techniques. 11
2.1.1. Configuration Management (CM) 11
2.1.2. Verification and Validation (V&V) 11
2.1.3. Quality Assurance (QA) 11
2.1.4. Standards 11
2.1.5. Project Management Plan 12
2.1.6. Software Safety Plan 12
2.1.7. Collection and Analysis of Metrics 12
2.1.8. Financial and Schedule Risk Analysis 12
2.1.9. Documentation Plan 12
2.2. Technical Techniques. 13
2.2.1. Safety Analysis of the Software System 13
2.2.2. Requirements Analysis Techniques 13
2.2.3. Design Techniques 14
2.2.4. Implementation Techniques 14
2.2.5. Testing Techniques 15
2.2.6. Formal Reviews and Walkthroughs 15
2.2.7. Proof of Correctness 16
2.2.8. Modeling 16
3. Thursday Morning Question-Answer Session 16
3.1. Common Mode Failure 16
3.2. Hardware Backup to Protect against Software Errors 17
3.3. Hardware versus Software 17
3.4. Certification of Programmers 18
3.5. Verifying Non-Safety Software 18
3.6. Numerical Rating of Software 18
3.7. Extrapolation from Measured Numbers 18
3.8. Guidelines on Hardware Versus Software Versus People 18
3.9. How Does Size Correlate with Complexity? 19
3.10. Partitioning the Design 20
3.11. FMEA and Fault Tree Analysis 20
3.12. Unintended Functions 20
3.13. Convincing a Regulator That a Design is Correct 20
3.14. Testing 21
3.15. Using a Safety Approach to Improve the Reliability Numbers 22
3.16. Final Remarks 23
4. Thursday Afternoon NRC / LLNL Discussion 24
4.1. General Points 24
4.2. Assessing Organizations 25
4.3. Methodology 25

Abbreviations

AECB	Atomic Energy Control Board
ALWR	Advanced Light Water Reactor
CASE	Computer Assisted Software Engineering
CM	Configuration Management
DARTS	Design Approach for Real Time Systems
DFV	Design for Validation
EPRI	Electric Power Research Institute
FMEA	Failure Modes and Effects Analysis
FMECA	Failure Modes, Effects and Criticality Analysis
HP	Hewlett Packard
HAZOP	Hazard and Operability Analysis
IEC	International Electrotechnical Commission
JPL	Jet Propulsion Laboratory
LLNL	Lawrence Livermore National Laboratory
LOC	Lines of Code
MilStd	Military Standard
MoD	Ministry of Defense
NASA	National Aeronautics and Space Administration
NPP	Nuclear Plant Protection
NRC	Nuclear Regulatory Commission
PE	Professional Engineer
PLA	Programmable Logic Array
PLC	Programmable Logic Controller
PSA	Problem Statement Analyzer
PSL	Problem Statement Language
QA	Quality Assurance
RTCA	Requirements and Technical Concepts for Aviation
SADT	Structural Analysis and Design Technique[1]
SAR	Safety Analysis Report
SEI	Software Engineering Institute
SREM	Software Requirements Engineering Methodology
V&V	Verification and Validation

[1] Trademark Softech, Inc.

Executive Summary

The Workshop on Developing Safe Software was held July 22-23 at the Hotel del Coronado, San Diego, California. The purpose of the workshop was to have four world experts discuss among themselves software safety issues which are of interest to the U. S. Nuclear Regulatory Commission (NRC). These issues concern the development of software systems for use in nuclear power plant protection systems. The workshop comprised four sessions. Wednesday morning, July 22, consisted of presentations from each of the four panel members. On Wednesday afternoon, the panel members went through a list of possible software development techniques and commented on them. The Thursday morning, July 23, session consisted of an extended discussion among the panel members and the observers from the NRC. A final session on Thursday afternoon consisted of a discussion among the NRC observers as to what was learned from the workshop.

The workshop was organized by the Lawrence Livermore National Laboratory (LLNL), Fission Energy and Systems Safety Program. The workshop moderator was Denise Cartledge of LLNL. The discussions were not recorded; instead, running notes were taken on a computer and continuously displayed at the front of the room. In this manner, errors in the notes could be noticed and corrected as they occurred. The notetaker was Dr. Lloyd Williams, Software Engineering Research.

The panel members were:
Mr. Ricky Butler, NASA, Langley, Virginia
Dr. Nancy Leveson, University of California, Irvine, California
Dr. Bev Littlewood, City University, London, England
Dr. John Rushby, Stanford Research Institute, Palo Alto, California

Observers from the Nuclear Regulatory Commission were:
Mr. Leo Beltracchi, Office of Nuclear Regulatory Research
Mr. John Gallagher, Office of Nuclear Reactor Regulation
Mr. Joe Joyce, Office of Nuclear Reactor Regulation
Mr. Hulbert Li, Office of Nuclear Reactor Regulation

Observers from the Lawrence Livermore National Laboratory, were:
Dr. Dennis Lawrence, Fission Energy and Systems Safety Program
Dr. Greg Suski, Fission Energy and Systems Safety Program
Dr. Robert Wyman, Fission Energy and Systems Safety Program
Dr. Lin Zucconi, Fission Energy and Systems Safety Program

The following report was written from the notes taken at the workshop and was reviewed by the panel members and observers for comment. It consists of the discussion notes and comments, and incorporates corrections by the panel members and observers. When there were disagreements among panel members, these disagreements are indicated in this report as appropriate. To facilitate a free and open discussion, it was agreed that comments other than the opening talks would not be attributed to specific individuals.

In an analysis of the workshop by the LLNL and NRC observers, a number of general conclusions were drawn as to the overall beliefs and recommendations of the panel members. These conclusions are summarized in the following numbered list, and represent the opinions of the LLNL and NRC observers, not necessarily those of any particular panel member.

1. General Conclusions
   a. Software safety requires a comprehensive approach, using more than one technique. This may include testing, formal development methods (including proofs of correctness), expert judgment and other factors.
   b. Safety and reliability are different, and both are important in reactor protection systems. It is not possible to just reduce the safety problem to standard correctness or reliability problems because safety is concerned with the *consequences* of system operation, not whether it meets its specifications (which may be incorrect).
   c. Software safety begins with a hazard analysis, which must be carried out at the system level. Software safety is a part of system safety.
   d. The designer needs to present a well-constructed argument as to why the system is correct, reliable and safe.

2. Conclusion on Diversity
    a. Diversity can be of value against common mode failures, but needs to be implemented correctly. Diversity should be introduced early in the design, at the highest level possible. Diversity, for example, could be achieved by using a software-based system and a parallel hardware-based system to satisfy the same requirement. In particular, n-version programming is not particularly useful.

3. Conclusion on Complexity
    a. There are two aspects of software complexity: functional and structural. The former is much more important to safety concerns than the latter, but the latter is emphasized by computer scientists because it appears easier to measure. Complexity is a problem because it may decrease the understanding of the software system.

4. Conclusions on Reliability and Testing
    a. Testing cannot be used to demonstrate the reliability of software beyond 10^{-4} failures per demand. Reactor protection software either must not require higher reliability than this, or the improvement in reliability must be derived using a method other than testing.
    b. Reliability is designed into a system and analyzed using bottom-up techniques. Safety is designed into a system and analyzed using top-down techniques.
    c. Reliability must be quantified to have any meaning. This is a difficult problem, but failure to calculate reliability frequently leads to excessive optimism about the safety and reliability of a system.
    d. Failure data should be collected on systems during actual operation, and then be analyzed. This is the only way the software community will eventually know what works and what doesn't.

5. Conclusions on Failure Modes.
    a. Formal methods are based on mathematics, such as predicate calculus, recursive function theory, programming language semantics and discrete mathematics. They vary according to the degree of rigor and coverage. Many forms of scrutiny are valuable even though they fall short of formal methods.
    b. Formal verification includes the following properties: specification of the system using languages based on mathematical logic, rigorous specification of desired properties as well as implementation details, and mathematical proof that the implementation meets the desired properties.
    c. The use of formal methods provides many benefits beyond correctness proofs. Formal methods make it possible to challenge or test software requirements, to expose assumptions, to manage the change process more reliably, to help keep system designs simple and to provide powerful tools for handling complexity where it is necessary. The biggest advantage to formal methods may be the intellectual rigor required, which can greatly increase one's understanding of the software.

1. Wednesday Morning Prepared Talks

The session opened with remarks by John Gallagher on the purpose of the workshop. Presentations by Nancy Leveson, Bev Littlewood, Ricky Butler and John Rushby followed, in that order. The main points of these presentations are given next. There was considerable interaction among the presenters, both during the workshop and during reviews of a draft of this report. In the description given here, comments from panel members are labeled "discussion" or "comment."

The discussions reported here contain many technical terms which are not defined in this report. The reader is assumed to be familiar with the general terminology used in discussions and writings about high reliability and safety-critical software systems. Another report is available which contains background information on software reliability and software safety.[2]

1.1. Talk by John Gallagher

The purpose of the workshop was to provide the Nuclear Regulatory Commission (NRC) with expert opinions on what methods and techniques are available to software developers that can improve the safety of software systems contained in nuclear reactor protection systems. The primary concerns are how the developer can avoid errors, find errors and mitigate the consequences of errors.

The NRC issues plant design licenses using a process known informally as One Step Licensing. Once a design is approved, it may be applied to constructing plants in various locations. The reactor vendor must

1. Perform a defense-in-depth analysis, preferably using NUREG 0493[3], and analytically demonstrate adequate diversity for each event evaluated in the accident analysis section of the Safety Analysis Report (SAR) for each postulated common mode failure.
2. Provide diversity where problems are identified within the digital protection systems. The developer may use a non-safety system to accomplish this.
3. Provide a hardwired non-computer-based manually operated actuation system as back-up for the computer-based protection system.

Currently the NRC is considering adopting an EPRI document[4], as the frame of reference for the establishment of design acceptance criteria.

NRC regulations should be tied directly to activities considered to be "good software engineering practice" to avoid charges of being capricious.

1.2. Talk by Nancy Leveson

There are two extreme positions with respect to the risk of technological systems.

1. Risk is a technological problem that can be solved by technological fixes and probabilistic analysis. This position is too optimistic. Accidents almost always have non-technological components; e.g., management mistakes. In some accidents, for example the Challenger accident, potentially effective technological risk reduction procedures were made ineffective by non-technological factors. History shows that attempts to attack safety problems using technology alone will usually fail and that probabilistic risk assessment often ignores the most important factors in real accidents.
2. Technological fixes to problems of risk are impossible and probabilistic risk assessment is useless. This position, espoused by people like Charles Perrow[5], is too pessimistic. There are many things that can be done to make engineered systems safer.

---

[2] J. Dennis Lawrence,, "Software Reliability and Safety in Nuclear Reactor Protection Systems: Interim Report," Lawrence Livermore National Laboratory (1992), in prep.

[3] "A Defense-in-Depth and Diversity Assessment of the RESAR-414 Integrated Protection System," NUREG-0493, Division of Systems Safety, Office of Nuclear Reactor Regulation, U.S. Nuclear Regulatory Commission (March 1979).

[4] *Advanced Light Water Reactor Utility Requirements Document, Volume III, ALWR Passive Plant, Chapter 10, Man-Machine Interface Systems,* Electric Power Research Institute, Palo Alto, CA (1990).

[5] Charles Perrow, *Normal Accidents: Living with High-Risk Technologies,* Basic Books (1984).

The answer lies somewhere in-between. Safety is a complex problem for which there is no simple technological solution. Managerial and social issues may be just as important in causing and preventing accidents. These factors cannot be measured probabilistically, but they need to be considered.

Alvin Weinberg[6] makes a convincing argument that there are questions that can be asked of science, but which cannot be answered by science. One of these questions is the measurement of risk. Probabilistic assessments can be useful for many purposes. The problem arises when we put too much faith in them and ignore the many assumptions underlying them. Ryder, one-time head of the British Health and Safety Executive, wrote: "The numbers game in risk assessment should only be played in private between consenting adults as it is so easy to be misinterpreted."[7]

Here are a few axioms about risk.

1. People tend to underestimate risk. As a result, complacency may be the most important root cause of accidents. This underestimation typically comes about due to people attempting to calculate the probability of an accident by assuming independence and multiplying probabilities of triggering events that are not really independent. Most accidents in well-designed systems involve two or more events of low probability occurring in the worst possible combination. The Brown's Ferry fire is an example of this in the nuclear power industry, but examples abound in nearly every type of dangerous system. After the fact, independent events often are found to have a common precipitator.

   Complacency about software is a common problem today. The Therac-25 accidents have increased awareness in the medical industry, but it may take an accident in each industry to get people to pay attention.

2. Upstream approaches to safety are the most effective and least costly. Safety features should be built into a system during development. This is not done today in most software systems. It is better to build in safety instead of adding on a protection system. For example, one can protect against fire by using smoke alarms and sprinklers, but then you are limited by the reliability of the protection devices. An alternative is to eliminate or minimize the potential for fire; for example, by using fire retardant materials or eliminating storage of flammable materials. The answer is not necessarily one or the other, but relying only on protection systems boxes you into a corner where your only choice is to try to achieve ultra-high reliability in these devices.

   In the nuclear industry, the over-emphasis on protection systems has been accompanied by an under-emphasis on the safety of the control systems. Thus, software in protection systems is classified as critical but usually that in control systems is not. Accidents have occurred as a result of this.

   Building in safety is not necessarily more expensive than adding protection devices. For years, manufacturers refused to do anything about the problem of children dying from suffocation inside refrigerators when they got locked in while playing. The government finally required them to do something, and magnetic latches were developed that can be opened from inside; they turned out to be less expensive than the older-style latches. An inherently safe process will often be cheaper than a hazardous one with many added-on protection devices and overdesign.

3. Software alone is neither safe nor unsafe. As a corollary, we cannot look at or measure software in isolation and determine whether it is safe. For example, in the Therac-25, a software flaw caused a massive radiation overdose in patients. In the Therac-20, the same software flaw resulted in blown fuses, but did not cause an accident or radiation overdose.[8] There are other examples of software being reused in a different system from that which it was developed and accidents or near-accidents were the result. Safety is a system problem. Individual components are not safe or unsafe. Assigning a "safety" number to a component is meaningless since the safety of the system depends upon the system within which the component is operating. A valve in a nuclear power plant is not safe or unsafe by itself, but it may be unsafe in certain plant designs. A reliability figure for that valve (probability of it failing) is not a safety figure for the valve. Even combining the reliability figures for all the components does not provide an estimate of safety since the problems most often arise from the interface; i.e., the behavior of the components individually is fine but is unsafe when put

---

[6] Alan Weinberg, "Science vs. Trans-Science", *Minerva*, vol. 10 (1972), 209-222.

[7] E. A. Ryder, "The Control of Major Hazards: The Advisory Committee's Third and Final Report," Transcript of Conference on European Major Hazards, Oyes Scientific & Technical Services and Authors, London (1984).

[8] Nancy G. Leveson and Clark S. Turner, "An Investigation of the THERAC-25 Accidents," UCI Technical Report # 92-108, Information and Computer Science Dept., University of California, Irvine (November 1992).

together. This required more than a simple ANDing of failure probabilities. Reliability is a bottom-up technique; safety must be top-down.

4. Software safety requires a comprehensive approach. Hazards must be evaluated at the system level and protection built-in at that level. The current trend to removing hardware interlocks and replacing them with a computer or controlling them with computers is dangerous and has led to accidents. After the hazards have been identified and protection included in the design at the system level, hazards must be traced to individual component behavior (including software) and protection built in against that particular unsafe behavior, if possible. Single methods (such as diversity) are not adequate to protect against accidents. The issue is how to put the methods together and design a comprehensive safety program.

The following are responses to questions posed by the NRC:

- There are two aspects to software complexity: functional and structural. Functional complexity is more important than structural complexity. The latter is easier to measure and identify, and therefore tends to get more emphasis, however. Measuring one aspect of complexity may just encourage designers to shift the complexity elsewhere. For example, attempting to minimize control flow complexity can result in an increase in data-structure complexity. Reducing intra-module complexity may require increasing inter-module complexity. In both cases, the second alternative may well be worse than the first. The real issue in safety is functional complexity, not structural complexity. From a system standpoint, complexity is often introduced when computers are introduced.
- Several new software standards have instituted assignment of criticality to components and then reduced the software development requirements (e.g., amount and type of testing) for lower-criticality components. This is not an effective approach. First, criticality is usually assigned on the basis of that component being able to cause an accident by itself. Since accidents are nearly always caused by the undesired behavior of two or more components (most systems are designed such that a single component cannot cause an accident), in practice this has led to almost no software components being declared as safety critical and to an increased probability that a lower-criticality component will cause an accident (since less care is taken in their development). Second, there is no evidence that any of the techniques mandated for any of the levels can be reduced without reducing drastically the reliability of the components. Most of the assignment of software techniques to levels is arbitrary and based on little or no scientific evidence of their efficacy. The system safety approach would instead be to identify system hazards, evaluate them with respect to severity (and perhaps likelihood, although this is very difficult and inaccurate before the system is designed and implemented), and then trace the hazards to the components. Any components that can contribute to unacceptable hazards are considered safety-critical, whether it is believed that they can cause the hazard alone or not.
- MIL-STD-882B is a good example of a safety standard.
- With regard to formal methods: Leveson's concern is the validation of the formal specification. Writing down the requirements in a formal language helps, but the specification may still be wrong. Specifications need to be formally validated as safe.
- It is incorrect simply to reduce the safety problem to a standard correctness or reliability problem. Software that is "correct" (i.e., satisfies its specification) and reliable may still be unsafe. Accidents have occurred even when software did not fail. Programmers must understand the system and software hazards in order to build protection into the software. Testers must perform special safety and stress testing for robustness and safety. Specifications need to be verified for safety as well as correctness.

1.3. Talk by Bev Littlewood

If we cannot assure ourselves that we have developed and built software to an ultra-high level of reliability, then we shouldn't build a system whose safety relies solely upon an ultra-high level of software reliability. There may be other ways of building systems to secure the advantages of software without compromising the safety of the overall system.

Two useful distinctions regarding failures are the following.

1. Failures due to design faults versus failures due to physical faults. This is not an issue of software versus hardware. Software suffers only design faults, but the problem is more severe due to the extra complexity of the software.

2. Methods of achieving reliability versus methods of assessing reliability.

Discussion: It was pointed out that safety and reliability are different, and should not be confused with one another. The distinction is important. Overemphasis on quantifying reliability leads to overloading, or ignoring things you can't quantify. The real villains are frequently uninformed managers. The technical people don't believe they can achieve 10^{-9} reliability, but nontechnical people frequently make such claims. There are limits to what you can say with numbers.

In response, it was stated that there are two issues here. One is the difference between safety and reliability, and the other concerns quantification. Reliability and safety each involve stochastic processes of events, and they are therefore very similar in terms of quantification. The events themselves are clearly not the same in the two cases, but the same mathematical approaches and the same language (frequency of failure, probability of failure on demand, and so forth) can be used in both cases.

Littlewood does not see how overemphasis on quantification leads to ignoring things that cannot be quantified. On the contrary, it has been the stochastic community that has been in the forefront of attempts to warn of the limitations to our means to gain confidence in system safety. In the United Kingdom, at least, the "gung ho" approach has come from those who have claimed that their (non-quantified) understanding was sufficient to banish uncertainty.

Those with a skepticism about quantification often end up placing a much greater confidence in systems than they would if they thought about the very difficult quantification issues arising from the nature of the evidence upon which their judgments are based. The whole point of some recent work is that the discipline of quantification makes us much more pessimistic. The reason is that the evidence that comes from sources that are not usually regarded as suitable for arriving at quantification of safety are in fact very much weaker than is often realized.

Software requirements specifications frequently state ridiculously high reliability numbers with no justification. We must then address the question of whether or not we can build systems that meet such required reliability levels. How can we convince ourselves that the required reliability has been achieved where software is involved?

Here are some examples of the "nature and extent of our dependence upon computers."

1. A320 flight control has a stated requirement of 10^{-9} failures per hour. Similarly for 747-400, 777, etc.
2. Sizewell B reactor protection requires 10^{-4} probability of failure upon demand.
3. Air traffic control requires 3 seconds downtime per year.
4. Chemical plants such as THORP have risks comparable to those in nuclear power plants.
5. Robotics (such as surgical assistance) have surprisingly modest reliability requirements.
6. Railway signaling and control require 10^{-12} probability of failure per hour.

Why do we need to express safety and reliability requirements in terms of probability?

1. There is an inherent uncertainty about the failure behavior of a system. This arises from the unpredictable nature of the operational environment and the observer's incomplete knowledge of possible system behavior.
2. Informally, we need to have sufficient confidence that the system will fail sufficiently infrequently. Alternately, for a one-shot system, we need to have confidence that the system will fail with sufficiently low probability.

Direct observation of operational behavior is not going to give assurance of ultra-high reliability.

1. One problem is the representativeness of input cases. How can you guarantee that the input observed actually matches the "real" input?
2. Another problem is the law of diminishing returns (plus issues such as "are your fixes fallible?"). After a while, additional testing yields little additional information about the reliability of the system, so is no longer practical.

Software reliability growth techniques are not applicable to ultra-reliability.

What parts of the software development process contribute to the reliability of the software product? There have been experiments in n-version programming which have shown that it results in some improvement, but not as much as naive theory, based on false assumptions of independence, might suggest. There is no empirical evidence for the efficacy of formal methods. To obtain ultra-reliable systems, we will need a composition of evidence from disparate sources - proof, statistics, judgment, etc.

This is a key *positive* point. It relates to "obtaining confidence" in ultra-high reliability, which can be termed the assessment problem, rather than "obtaining the reliability."

Discussion: It was pointed out that obtaining confidence requires Baysian prior belief. We can't quantify 10^{-9} failures per hour. We can quantify 10^{-3} failures per hour and give arguments as to why we think the actual probability is higher. This amounts to engineering judgment, which is quite valuable when based on actual knowledge and understanding. On the other hand, when we look at such judgments quantitatively, they're surprisingly bad. We shouldn't trust judgments too much - we need to formalize them. Consider what the lack of experience on the part of an engineering group may produce when asked to exercise their engineering judgment. Each generation of safety system developers builds only one system, so they don't gain the experience needed upon which to base engineering judgment.

In further discussion, it was stated that you can't separate a control system from its context. This implies that you should keep the requirements simple. Conflicting requirements, such as performance and economics, lead to complex systems. Finally, the reliability of control systems, interpreted narrowly, is not the central issue. The central issue is the reliability of the *entire* system.

In a later response, a panel member stated that the point of this discussion is not clear. The last sentence of the preceding comment is true, but it is not clear how it follows from the first part of the comment. Are we talking about reactors here? That is, are we talking about a system design where there is one system for everyday control and a second protection system for safety when the control system allows the reactor to get into a dangerous state.

People tend to err in two different ways. In the first place, they tend to be far too optimistic. For example, they may be too optimistic about the reliability of a system they have built. Secondly, they are far too optimistic about their own propensity to err; they will tell you (and believe it strongly) that their belief in the correctness of their judgment is greater than is warranted by the facts. There is a nice example in some work by Henrion and Fischhoff[9] about the errors made by physicists over the last century in measurements of constants such as the velocity of light, the charge on the electron, etc. In all cases, it was interesting that not only were they quite spectacularly wrong when compared against present knowledge, but their confidence was misplaced; their confidence intervals for the true value (taking account of their own judgments of error) often did not contain the modern value.

Here is a list of areas where technical work is required. I am pessimistic about the outcome.

1. Real statistical evidence for the relationship between processes and product attributes.
2. Formal quantification of expert judgment, combination of expert judgments, calibration, etc.
3. Composition of evidence from disparate sources, such as proof, statistics, judgment, and others.
4. Quantitative theory to replace the present qualitative "claim limits."
5. Better probabilistic modeling of "structured" software. Diversity, fault-tolerance, voters, failure clustering, etc.
6. Accelerated testing.
7. Better data: experiments, case studies and mandatory reporting requirements for safety-critical systems in operational use.
8. Standards: better scientific basis, studies of efficacy, and so forth.

In actual practice, safety systems are not isolated from other systems, and they should be.

How can we tell what using a formal method has delivered to you? We lack empirical evidence of this.

Requirements for critical systems should involve probability-based requirements for subsystems, including software. For each critical system, it must be demonstrated that the required level of reliability has been achieved. Critical systems with requirements that cannot be validated should not be certified for use, and should not be built.

Some applications appear to need reliability levels that are orders of magnitude higher than we presently can assure. In some cases, the required level will never be achievable. By proper system design, however, we may not need such high levels. For example,

1. Use provable safety kernels with "add-on" goodies.
2. Reuse of software with high assured reliability from long operational use.

1.4. Talk by Ricky Butler

Avionics safety systems are different from process control safety systems because you can't have a separate shut-down system in an airplane. The control system has to fly the plane. Here, we are talking about ultra-high reliability in the 10^{-9} area.

[9] M. Henrion, B. Fischhoff, "Assessing uncertainty in physical constants," *American J. of Physics*, vol. 54, 9 (1986), 791-798.

Earlier airplanes had a mechanical linkage from pilot to control surfaces. Then there was a change to hydraulic linkages. Now military aircraft use electronic linkages. Commercial aircraft are going that way now.

NASA Langley has a research project on "design for validation" (DFV)[10]. The driving factor of this philosophy is the need to produce a credible reliability number. DFV has the following components:

1. The system must be designed so that a complete and accurate reliability model can be constructed. All parameters that cannot be deduced must be measurable in feasible time under test.
2. The reliability model does not include transitions representing design faults; analytical arguments must be presented to show that design faults cannot cause system failure.
3. The reliability model must be shown analytically to be accurate with respect to the system implementation.
4. Design tradeoffs are made in favor of designs that minimize the number of parameters that must be measured, and that simplify the analytic arguments.

The DFV philosophy is to design so that there is no single-point failure. This is demonstrated by a formal proof. Analysis is then used to avoid the need for fault-injection experiments. The DFV philosophy rules out digital design diversity as a means of primary assurance of safety because its efficacy cannot be demonstrated. The independence assumption is fundamental to reliability modeling of fault tolerant strategies.

Discussion: It was pointed out that the difficulty is to ensure that there are no common physical failure modes. The reliability analysis should be done as if there were independence in the arrival of physical faults in electrically-isolated channels. Everything should be done to eliminate common-mode failures. This strategy is justifiable for physical failures but not for design errors.

Empirical evidence shows that even low reliability systems do not exhibit independence of the manifestation of design error in multi-version software / hardware. This does not say that design diversity shouldn't be used. It only says that the numbers that can be obtained using models that assume independent error manifestations in the multiple versions shouldn't be relied upon.

Discussion: The problem is not solved by using different programming languages. Languages are not the source of failure, the failure is due to the difficulty of the problem. The independence assumption can never be shown to hold for ultra-reliable software. Verifying independence requires running the system for centuries. If the goal is more modest - say, 10^{-4} - there are some ways to quantify independence. However, if "diversity" means having diverse failure modes, this can't be guaranteed for software. For example, the use of threshold voting increases complexity and could lead to errors. Deciding how to set thresholds is very difficult. There may be diversity in hardware, software and time, as is being done in the Boeing 777. This system uses nine-fold redundancy: three asynchronous channels, each channel with three lanes that utilize dissimilar hardware and software. The amount of complexity here is staggering.

Diversity is not free. Using diversity requires a trade-off of complexity in the voting algorithm. The question is: what risk do you add with diversity? Diversity management can itself become the major source of failure.

Regarding design diversity, it is not possible to extrapolate from a low reliability system to a high reliability system. Measuring coincident errors in systems with diverse designs will give at best (due to testing time limitations) 10^{-4}. If the goal is 10^{-4} or 10^{-5}, then a lot of time must be spent testing. Note that in software, exact match voting similar to what can be done with hardware is not possible. Software must use some sort of threshold voter. This will add the complexity of the voter into the reliability equation.

Controlled experiments can be done to get evidence on use/payoff of formal methods but this will only help in the low reliability regime. The alternative is the basic understanding gained, which increases confidence in the system.

A control system should have predictable, deterministic timing properties. Fault tolerance overhead should be a small fraction of total system overhead.

Simplicity: The fault-tolerance and redundancy management portions of the system should be transparent to the applications programmers. The delivered system should require very little (or no) maintenance. This reduces costs and increases safety. Many failures are maintenance-induced, especially unplanned maintenance at marginal facilities.

Here is a list of characteristics of formal verification.

1. Specification of the system using languages based on mathematical logic.

---

[10] Sally C. Johnson and Ricky W. Butler, "Design for validation," *IEEE Aerospace and Electronic Systems J.*, vol. 7, 1 (January 1992), 38-43.

2. Rigorous specification of desired properties as well as implementation details.
3. Mathematical proof that the implementation meets the desired abstract properties.
4. Use of semi-automatic theorem provers to ensure the correctness of the proofs.

In principle, formal methods can accomplish the equivalent of exhaustive testing.

Formal methods are based on mathematics.

1. Consistently successful engineering of complex computing systems will require the application of mathematically based analysis analogous to the structural analysis performed before a bridge or airplane wing is built.
2. The mathematics for such analysis is logic, just as calculus and differential equations are the mathematical tools used in other engineering fields.
3. The mathematics of formal methods includes: predicate calculus (first order logic), recursive function theory, lambda calculus, programming language semantics, and discrete mathematics (number theory, abstract algebra).

There are several levels of formal methods.

1. Static code analysis, no semantic analysis.
2. Specification using mathematical logic or language with a formal semantics; that is, meaning expressible in logic.
3. Formal specification plus hand proofs.
4. Formal specification plus mechanical proofs.

European emphasis is on level 1, while US emphasis is on level 3 (due to a large NASA investment). The Europeans are ahead in the transfer of this technology to industry, while the US is ahead in tools for formal verification.

There are three basic approaches to achieving reliability.

1. Testing - lots of it!
2. Software fault tolerance
3. Fault avoidance, by formal specification and verification, automatic program synthesis and reusable modules.

General conclusions:

1. Direct life testing of software is not feasible for the ultra-reliable region.
2. Reliability growth models do not significantly decrease the test time.
3. The independence model cannot be assumed for fault-tolerant software.
4. No coincident-error model can be experimentally validated for the ultra-reliable region.

1.5. Talk by John Rushby

Experience in the field is not transferred into new projects. For example, the way to build space station software was not learned from the shuttle software.

Redundancy management can get out of control, and can be the primary source of failure.

DO 178B[11] (draft 6) from RTCA for flight control includes the following.

1. The level of reliability cannot be measured after delivery (that is, by testing), so it must be assured by means of an appropriate process.

   Comment from another panel member: "I think it is much worse - more dangerous - than this. My paraphrase of the DO 178B approach would be 'we can't measure, so we are not going to try, but we shall nevertheless claim 10^{-5} because we have used good practice.' It is dangerous, unscientific rubbish."
2. Use stress hazard analysis, with reviews by independent teams.
3. There is malfunction, unintended function and loss of function. These are different.

   Comment from another panel member: True, but these are merely events when we wish to evaluate safety or reliability - albeit with differing severities.
4. Design methods include:
   - Partitioning (fault containment)
   - Protection (least recommended because it comes late),

[11] "Software Considerations in Airborne Systems and Equipment Certification," DO-178B Draft 6.0, Proposed Revision to DO-178A, Requirements and Technical Concepts for Aviation, Washington, D.C., Working Paper (April 28, 1992).

- Dissimilarity (should only be used to achieve additional confidence after the primary requirements).

5. Distinguish between review, which is a human consensus activity, and analysis, which is a repeatable method.
6. Fagan style inspections are very useful. The big win is early in the life cycle. (JPL found that by doing Fagan inspections they found one serious error in every three pages of requirements documents. Two-thirds of these were omissions. Code inspections found one error per twenty pages, since the errors were found earlier.)

There are dangers in using design diversity, such as n-version programming and recovery blocks.

1. It creates an illusion of ultra-reliability. By assuming independence, the advocates of software fault-tolerance generate ultra-high estimates of reliability.
2. As long as industry and certification agencies believe that software fault-tolerance will solve the problem, formal methods will not be pursued.

NASA accepts that the level of reliability they want can't be measured. Therefore, the emphasis is on the development process.

Another panel member commented that this seems to imply that the development process can deliver something beyond what is measurable. If so, how would we know that we had achieved the desired reliability? This particular panel member stated that this is the issue upon which "I might differ most strongly from the others. It seems to me that we must always evaluate, even if we only do this in terms of 'engineering judgment' or something similarly informal. I just do not believe that we could come up with convincing arguments for much stronger belief than is represented by what comes from a more formal measurement via, say, direct observation of failure behavior. In any case, any claims to be able to do this must be justified by those who make them; I would be interested to hear how this could be done, even in principle."

The context for software evaluation is hazard analysis. For the most critical software, very elaborate reviews are employed. These increase scrutiny on the process.

Discussion: There is no scientific evidence that there is a correlation between process and product. A hazard cannot be identified with a specific software component since hazards relate to the entire system. The context for criticality is hazard analysis. For example, one airplane had all of its failures in the redundancy management portion of the software. You should also look at Mil-Std 882B which takes a different approach.

Many forms of scrutiny are short of formal methods, but are still valuable. DO 178B describes the distinction between reviews and analysis.

Formal methods cannot be injected into a chaotic process.

Proof of correctness is the least valuable aspect of formal methods.

Formal methods can help with validation by

1. Making it possible to challenge or test the requirements by posing hypotheses and proving theorems.
2. Exposing assumptions. Proving a theorem about a specification requires assumptions; what are they?
3. Managing change more reliably. The machine can help manage the process. Errors creep in with change.
4. Encouraging simplicity because formal methods are hard to use.
5. Providing powerful tools for handling complexity where it is necessary.

Discussion: Not enough emphasis is placed on validation of formal specifications. Formal methods can win by helping to find errors or inconsistencies; they can also lose since formal methods are less understandable to domain experts. The value of formal methods is the ability to do analysis (including simulation). Note that there is no empirical evidence for this, but the panel members believe it.

Simulation isn't that great because control systems are too complex. There is also the problem of designing test cases for the simulation which mimic the later reality of the operational system.

One problem is how few people there are with experience building these systems (for example, space probes). Lots of mistakes, such as using too much complexity, are made over and over again.

Controlled experiments (to demonstrate the effectiveness of formal methods or other techniques) are difficult and expensive. They might be do-able for low (10^{-4}) reliability, but the results can't be extrapolated. Using formal methods increases the understanding and therefore confidence in the system. The danger is that use of formal methods becomes more of a checklist and the benefits are not realized.

Another panel member commented that without empirical evidence, such confidence in formal methods may be misplaced. This was referred to as the "it has to be right because I used mathematics" argument!

2. Wednesday Afternoon Discussion

As part of the preparation for the workshop, the panel members were sent a list of possible development techniques, generated from a variety of sources and presented with no assumption about their absolute or relative value. The panel was asked to discuss which techniques might be useful in preventing, detecting and mitigating errors. The panel decided that this approach was not useful to them, and that they should (instead) just go through the list and comment on the items. In many cases, little was said about an item since the panel members merely assumed that everyone would do it. In other cases, there was extensive discussion. The notes here list the items discussed, with comments on the discussion as appropriate. Lack of comments here merely implies that the panel members had little or nothing to say about an item. Inconsistencies in the comments reflect disagreements among the panel members.

Techniques are divided roughly into two areas, Management Techniques and Technical Techniques.

2.1. Management Techniques.

2.1.1. Configuration Management (CM)

This is a standard technique that should be used. CM is a well-developed field, and there are several good tools available. DO 178A and other standards are applicable. Safety critical systems should not be highly reconfigurable since we can't certify the safety of individual components independent of the system in which they will be used. The developer must decide when to redesign a system instead of modifying the current system design.

2.1.2. Verification and Validation (V&V)

The certifying agency should review all techniques (for example, formal methods) in depth. If the agency doesn't have the expertise, then an independent V&V team that does have the expertise should be used. The verification team should be independent of the development organization. If a new technique, such as formal methods, is used by the development organization, the organization will have to decide which one(s) to use, and how to implement them. If the development organization is not familiar with the new method, there will be a technology transfer issue. Finally, management needs good criteria for selecting the new method.

2.1.3. Quality Assurance (QA)

QA needs to be more than just filling out checklists. What are the QA criteria and guidelines for software, especially safety critical software? Note that developing software is different from manufacturing hardware products, so the QA techniques will be different. One question is where the dividing line between V&V and QA falls? One way of looking at this is to say that V&V is part of QA. Victor Basili says that quality assurance and quality control are different things. A final issue is distinguishing QA in the abstract from QA in practice as it will apply to a well defined, known effort, product or project.

2.1.4. Standards

Care needs to be exercised in choosing the standards to be used since some standards are very lax. Since there are lots of standards, this can be a difficult process. Good standards state what must be done (and not done), not how it must be done. The implementing organization must specify the “how.” The choice of standards must be approved by the regulatory body. Examples of standards include: Mil-Std 882B, IEC-880[12], MoD-00-55[13], and MoD-00-56[14]. Mil-Std 882B is a good choice because it says what

[12] “Software for Computers in the Safety Systems of Nuclear Power Stations,” IEC Publication 880, International Electrotechnical Commission (1986).

[13] “The Procurement of Safety Critical Software in Defence Equipment,” Interim Defence Standard 00-55, Ministry of Defence, Glasgow (April 5, 1991).

must be done, not how, and leaves it up to the developers to convince the reviewer that they have complied with it.

One panel member did not feel that MoD-00-56 was an example of a good standard. This member prefers the system safety engineering approach to risk, as contained in Mil-Std-882B. MoD-00-56 takes a more reliability-oriented approach to risk. MoD-00-55 also is limited in its proposed approaches to safety-critical software.

There are very few criteria available to use in choosing between the various standards. It's also difficult to measure compliance with a standard. Are certain practices forbidden or is there a required demonstration that certain criteria have been met? (This is the difference between "what" and "how" standards.) Standards are often used as guidelines, with specific implementations negotiated between the regulator and the developer. How is it determined that the standards are implemented with goodwill?

2.1.5. Project Management Plan

The only comment was “have one!”

2.1.6. Software Safety Plan

It needs to be an integral part of the system safety plan. System safety plan writers often write a software requirement which the software people haven't bought into, which can be a real problem.

2.1.7. Collection and Analysis of Metrics

Data should be collected on a development project; the data to be collected should be predicated on the project goals. Data collection should continue after the product is in place. DOD requirements for data collection are good, especially with respect to safety.

What is actually measured? What metrics demonstrate that the Project Management Plan is actually being followed?

There is a difference between product metrics and process metrics. Metrics are needed to indicate whether non-functional requirements are on schedule.

Complexity metrics [as they now exist] are mostly “snake oil.”

The collection of data on an operational product cannot be emphasized too much. It is a scandal that there are no mandatory reporting requirements for voting errors (vote-outs) in fault tolerant systems in civil avionics, for example. It is known, for example, that vote-outs have occurred in the A300 and A310 series, but there are no trustworthy statistics.

Given the enormous expense of experiments, it is unlikely that we are going to see many of them, so collecting real-life data is the only way the community is going to learn. Real-life data is also likely to be more realistic than experiments would be.

2.1.8. Financial and Schedule Risk Analysis

The use of formal methods requires that more time be spent before code is generated; this in turn requires management commitment. Risk analysis helps get management aware of budget and resources need to deal with risky issues. For example, formal methods requires a significant commitment of funds up front.

2.1.9. Documentation Plan

The primary comment was to have one! Project costs are strongly dependent on the quality of the documentation. How can a regulator distinguish between a good faith effort and pro forma documentation? There is a need to have a detailed analysis to verify the documentation. Someone must sign off on the analysis results. A final question is: how is the degree of compliance measured?

---

[14] “Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment,” Interim Defence Standard 00-56, Ministry of Defence, Glasgow (1989).

2.2. Technical Techniques.

2.2.1. Safety Analysis of the Software System

The following examples were given in the methods list:

FMEA, FMECA.
Fault Tree Analysis.
Event Tree Analysis.
Hazard Analysis, HAZOP.
Risk Analysis.
Probabilistic Risk Assessment.
Reliability Block Diagram.
Sneak Circuit Analysis.
Human Factors Analysis.

The need is to present a convincing argument that all foreseeable problems have been covered. The safety view is concerned only with hazards, while the reliability view is concerned with all faults. FMEA and FMECA apply to system architecture, and do not really apply to software. The FMEA can specify what the software can do to address system hazards and to identify areas where software can contribute to or add hazards. FMECA adds consequences to a FMEA. These techniques are needed to tell what the software needs to do to protect against hazards or what the software can do to help recover from failures.

Fault Tree Analysis can be used on software. Sneak Circuit Analysis is misapplied to software; don't waste the time. It is done by converting software into circuit diagrams, and then looking for sneak circuits with proprietary rules. Nothing real is found. It's really standard static source code analysis. There are much better tools (for example MALPAS).

Human Factors Analysis is used to determine what kinds of roles are reasonable to assign to people, software, and non-computer hardware systems. How are humans provided with information so that they are not forced into errors? How is it shown that everything has been considered?

It is easier to provide fault tolerance against identified faults than to provide robustness for all possible faults. The issue of adding complexity involves trying to handle many identified faults versus building with overall robustness. An example of the latter is a robust system architecture. A question is: how is it shown that there is total coverage? Unsafe operation can occur without a "failure."

2.2.2. Requirements Analysis Techniques

The following examples were given in the methods list:

Object Oriented Requirements Specification.
Data Flow Requirements Specification and Analysis.
Requirements Tools - SADT, SREM, PSL/PSA, etc.
Use of CASE Tools.
Requirements Safety Analysis.
Software Hazard Analysis.
Defense in Depth Analysis.

The techniques listed here express requirements in terms of the mechanisms for achieving them, but they have limitations. Specific techniques should not be required. Formal methods are more sophisticated because they allow specification in terms of properties rather than mechanisms.

Regarding Defense in Depth Analysis: if the primary system fails, what exists to serve as a backup? If the control system is used as back-up, it is no longer non-critical. In general, this may mean that a non-critical system becomes a critical one.

Independence is a problem. If the control system gets the system into a critical situation, what is the probability that the shutdown system will not be able to respond properly? This is a global system issue, not just a software issue.

The issue of independence is a fundamental one throughout this whole technical area. Whenever independence cannot plausibly be assumed, quantitative claims for reliability or safety cannot be made.

There are no good ways of reasoning about dependence. Are there any ways the formalists can help here? This is one of the most important research problems associated with “evaluating” reliability and safety.

2.2.3. Design Techniques

The following examples were given in the methods list:

Semi-Formal Design - Data Flow Diagrams, Warnier Diagrams, etc.
Real Time Design Techniques - Boeing-Hatley, Ward-Mellor, DARTS, etc.
Object-Oriented Analysis and Design.
Formal Design Methods.
Use of CASE Tool.
Graphical Design Tools.
Tabular Design Tools.
Program Design Language.
Formal Design Review.
Use of Watchdog Timers.
n-Version Programming.
Recovery Blocks.
Design Safety Analysis.

The panel suggested some questions that need to be asked about design techniques: Does the technique fit the problem, and are the people who are using it familiar with it? The question for the NRC is: why did the developer choose the technique? Is the technique helpful, is it necessary, is it sufficient? Some may be helpful; none is sufficient.

Does the organization have the ability to use the techniques chosen? Are the techniques chosen sufficient to deliver the degree of safety and reliability that is required? Do the techniques match the level of risk we're willing to live with?

The problem is that there is very little hard scientific evidence for the efficacy of *any* software engineering methods or techniques. It is not likely that the last two questions can actually be answered.

CASE tools are really just picture drawers at present.

Formal design reviews are helpful.

The principal requirement is that there must exist a well constructed argument as to why the system is correct and safe. The question is: which of these techniques address this issue and which don't? For example, CASE doesn't address the issue, but formal design reviews do address it. Others (such as watchdog timers) are specific techniques to address specific problems.

Proponents of n-version programming have made unjustified claims. The technique does not give independence, but it seems to provide some benefit, and other techniques on the list are probably subject to the same criticisms. N-version programming is one of several techniques that can be used, but it must be used carefully and analyzed carefully. N-version programming only protects against certain kinds of errors that are easily tested for. In particular, it doesn't protect against requirements flaws. The problem is achieving independence and knowing that it has been achieved. There may be other ways of achieving diversity, but they all have similar problems.

N-version programming can't be applied to some problems (for example, clock synchronization); some parts of the program will be global across the redundancy. Unfortunately, these are often the most critical.

Formal methods help by making assumptions explicit and by helping to direct the testing and validation process. They are not complete solutions but they limit and define the incompleteness.

The evidence for effectiveness of all of the techniques listed is anecdotal.

2.2.4. Implementation Techniques

The following examples were given in the methods list:

Language Choice. (Which?)
Mandating or Forbidding Certain Language Statements. (Which?)
Coding Style.
Walkthroughs.
Code Inspection.

Formal Code Review.
Code Safety Analysis.

It is better to state a goal and require a convincing argument that the goal has been achieved rather than specifying how to the goal is to be achieved.

Some languages provide some protection from errors (for example, strong typing) while others (for example, the C language) are prone to certain kinds of errors. Language choice is a secondary issue; the compiler is more important!

Late lifecycle problems are well-understood and there are techniques for dealing with them. Early lifecycle problems are harder. The problem is that people put more effort on late lifecycle problems because they understand them better.

Synthesis (transformational implementation) is practical for some domains. But it doesn't solve the whole problem because the errors are most often introduced in the specifications, not the translation into code.

There was an attempt in one case to get certification on the language translator and then to omit certifying the application program.

Regarding program synthesis: an error in the equations is more likely than an error in the program synthesis (translating detail design to code.)

It is important to distinguish between trying to find problems in a program and evaluating (showing the goodness of) the program.

Continue the reliability analysis into the early operating stage. Continue review analysis until there is enough data and the right kind of data.

2.2.5. Testing Techniques

The following examples were given in the methods list:

Static Code Analysis.
Dynamic Code Analysis.
White Box Testing.
Black Box Testing.
Stress Testing.
Symbolic Execution.
Statistical Testing.

Do it! [Testing.]

All of the listed techniques except statistical testing are aimed at improving the product. Statistical testing is aimed at finding out those coding errors which cause problems in actual usage. There is some evidence that statistical testing does help in assessing reliability, at least in the 10^{-4} range. Statistical testing augmented with formal arguments may be convincing of high reliability.

The problem is that there are no rigorous ways of doing statistical testing. The combination of logical and probabilistic information remains an unsolved problem.

It is very difficult to use tests to simulate real-life situations. Fixing errors resets the statistical assumptions. It would be a good idea to collect data during operation, perhaps with a provisional license.

Statistical testing should be done even it only delivers the 10^{-4} number, but combine it with other techniques.

2.2.6. Formal Reviews and Walkthroughs

There was little discussion of this. Fagan inspections were somewhat endorsed.

2.2.7. Proof of Correctness

It is difficult to do this on code. Other techniques for code are effective and the bigger win with formal methods is in the early stages of the lifecycle. Apply proof of correctness to judiciously chosen pieces.

2.2.8. Modeling

There are two types of model. Finite state models combined with animation or simulation are useful early in the lifecycle. Reliability models are different, and have a different purpose.

The foundations of reliability models (especially Markov models) need to be scrutinized carefully. Published comparisons on existing modeling codes may not be reliable.

For modest reliability, reliability growth models are often useful.

Various reliability modeling tools exist, but many of them are not useful. A consensus of the panel members was that comparisons in the literature have been self-serving. Their judgment is not to use modeling tools blindly but to look into each one carefully and understand how it works and what its limitations are.

One question that must be resolved in practice is when to move from a reliability growth model, in which errors are corrected, to a straight reliability model, which permits no more repairs.

3. Thursday Morning Question-Answer Session

This session consisted of questions from the observers and responses from the panel members, plus a few comments from the observers. This portion of the workshop report lists the questions and the various responses. Many of the questions resulted in a wide-ranging discussion, and there were disagreements among the panel members on answers to questions. We attempted to capture all this in these notes.

3.1. Common Mode Failure

Is there a technique to analyze for common mode failure? How can you mitigate the effects of common mode failures?

- How does a common mode failure differ from other problems (for example, "bugs")?
- To be able to demonstrate diversity, you need to show that two components perform different functions – this is as hard as proofs.
- The real problem is having the same conceptual error in two components that are supposed to be different.
- If you have the same code in different channels, you have a common mode. One solution is to not put everything in software. Even with different code (n-version programming) the failure modes are not completely independent. The issue is not software versus hardware. What is needed is some alternative means that is conceptually very different from the software solution. It may be that implementing in hardware is a way out, but it is because of the incidental benefit you get from being forced to think of a novel solution that *can* be implemented in hardware.
- Is there a "structured analysis" technique? No. The only way to handle the problem is to perform verification. (Another panel member doesn’t see how verification tells us anything about common modes.) Introducing diversity late in the lifecycle is not effective. It is better to get it in as early as possible. For example, diversity can be added by using different designs rather than different implementations.
- One problem is that you can use different designs but then they may end up on the same type of processor with the same operating system.
- If you have a complex software system with a simpler hardware secondary system, you might be able to prove the correctness of the secondary system. Then you could postulate that the failure modes are independent because the secondary system has been proven correct and will fail only for hardware reasons.
- In n-version programming, even when the programmers implement different algorithms to solve the same problem, they tend to make mistakes handling the same difficult inputs (e.g., handling boundary conditions).
- If you do a complete formal approach and prove everything, then you have a proven correct program and don't need redundancy. But even then you can't rule out a specification error.
- The only way to get a handle on assurance with respect to common mode failure is with correctness proofs. Also diversity will be most effective if diversity is at the highest level.

- Suppose you use functional diversity but then use identical microprocessors with same operating system? What is the effect?
- Can you achieve diversity by putting it in both hardware and software?
- The ultimate level of defense is the operator.
- An issue is the situation in which hardware has some software or microcode in it which could be an unknown.
- It is not clear how to assess the true effectiveness of various ways of implementing diversity.
- Convincing, complete logical arguments are needed for the correctness of applications of PLCs and PLAs, no matter what the implementations. These are merely computers and are likely to suffer from all the problems we have been discussing. At the very least, we should be reluctant to accept any arguments that say that there is no need to worry about software.

3.2. Hardware Backup to Protect against Software Errors

Is it fair to conclude that having some hardware backup to protect against common software errors is good?

- Having hardware backup is better but there can still be errors.
- The issue is simple backup, not hardware backup. The point is that the backup should be sufficiently simple that its correctness can be reasoned about. It is possible this could be done in software if it were simple enough.
- You also have to look at functional complexity.
- There are no measures of functional complexity. The point is that we want intellectual control. Maybe we can simplify the software system and reduce the number of states. Building to a higher level abstraction may reduce the complexity. By using a higher level of abstraction, the number of states is reduced.
- Use deterministic rather than non-deterministic methods. For example, interrupts versus polling comes down to reducing the number of states. Use a state model - we can get control of a design when we can build a deterministic state model and can understand it.
- Complexity is related to the depth of the theorems that prove the correctness.
- From the view of regulators, a zero or one measure of safety is needed. That is, the system is either safe or it is not. Another panel member disagreed that the issue is so black and white.
- There is no handle on functional complexity.

3.3. Hardware versus Software

Is hardware simpler and easier to understand than software?

- It may be easier to understand or verify hardware than software.
- Hardware is not any easier to understand if the functionality is of the same extensiveness in the two cases. Isn’t this all a problem of complexity rather than hardware versus software?
- Using hardware to back up software may be problematic due to the translation from hardware (the current system) to software and then back to hardware.
- If you're going for diversity, make it as diverse as possible. Hardware is different from software.
- The distinction between hardware and software is fuzzy (cf., PLCs).The diversity may not be as great since the problems are still intellectual (programming).
- There is a spectrum from hardware only to a mixture of hardware and software, with PLCs in between.
- The key is having a way to verify it.
- There may be some advantage to using a PLA or PLC but the advantage is not in just using it. The advantage would be if you could construct a more straightforward argument for correctness of the design if a PLA or PLC is used. What you should look for is convincing and complete logical arguments for correctness. There are no shortcuts – you have to pay the price of understanding it.

3.4. Certification of Programmers

Should programmers be certified?

- This is a political impossibility. You should have someone like a professional engineer (PE) to sign off on the project and accept responsibility for its safety.

3.5. Verifying Non-Safety Software

Should you be as thorough in verifying the non-safety software as you are in verifying the safety software?

- Historically, it has been possible to separate safety and non-safety systems. At present it's more difficult because the control system may have a role in responding to safety events.
- You've got to expand your viewpoint. Correctness is not the only criteria, and if you buy off-the-shelf software, you have to perform a full safety analysis since safety is a system property.
- The Ontario Hydro Darlington plant shutdown software was designed so that even if there were still residual errors of certain types that caused the software not to provide an answer or if there were certain types of control flow errors in the code that caused it not to execute all the necessary routines, the protection system would still shut down the plant.
- You need to quantify the problem. You don't need the system to be completely safe, but it has to be "safe enough".
- There is no way to answer the question of whether something is “safe enough.” This is not a question that can be answered using mathematics or science. It is dangerous to pretend you can quantify something when it cannot be quantified.

3.6. Numerical Rating of Software

Numbers are important for finding out precisely what is known. They can be used for relative rankings. Hardware was easier to quantify. Does anybody believe that you can do better than 10^{-4} with software?

- The number depends on the context (how many test cases). For reactors, 10^{-4} is probably on the boundary of what you can measure. You may be able to achieve better but it will be hard to demonstrate.
- Even if we get empirical data in the 10^{-3} and 10^{-4} region, getting 10^{-7} and extrapolating to 10^{-9} and higher is unrealistic. We can't get better numbers.
- There is still controversy about whether numbers which are meaningful or accurate can be obtained for software.

3.7. Extrapolation from Measured Numbers

Can you extrapolate from numbers such as 10^{-4} to numbers such as 10^{-9}?

- Extrapolation may be possible in the low reliability region but is dangerous for high-reliability.
- We just don't know about the high-reliability region – the types of errors may be different.

3.8. Guidelines on Hardware Versus Software Versus People

Can you offer any guidelines for hardware versus software versus people?

- One aspect is functional complexity versus structural complexity. Aspects of complexity include understanding, and the number of failure modes. Hardware may be simpler because it has fewer failure modes. Deterministic systems may have fewer states and be easier to understand. Polling systems may have fewer states. It's good to have control over the states that the system can take and be able to understand them.
- Complexity is related to the intellectual content; to the "depth of the theorem." Complexity is related to the size of the proof that you have to construct to demonstrate correctness. Other measures are worthless. (Side comment: is it really possible to measure the “size” of a theorem?)
- Structural complexity may be high but produce a lower functional complexity. For example, you might need to design a system with high structural complexity in order to achieve low functional complexity.

- The only useful measure of complexity is a binary one. It is either sufficiently simple that we can make a plausible claim that it is completely correct, or it is not and we are back dealing with uncertainty and probability. If we do not have complete certainty in correctness, nothing can be claimed from a purely logical perspective, at least if we are interested in evaluating how well the system will behave with respect to safety or reliability.
- You have to postulate a system that will work when other parts have broken. This is why people want diversity.
- People want to correlate complexity with lines of code. There is no such correlation.
- You don't have to evaluate all of the possible ways a system can fail, and you probably can't. Fault trees work backward from the hazard and are better than FMEA.
- There was a discussion on the role of the control system versus the protection system. A comment was: the distinction has been blurred in some nuclear plant protection (NPP) software, violating basic engineering design principles. This may have unfortunate consequences. We should not be ignoring lessons learned about engineering design when we introduce computers.
- In the nuclear area, at least in the United Kingdom, there is no blurring between control and protection. The protection and control roles seem to be clear and the allocation of numerical requirements to them in the overall safety case appears reasonable. On the other hand, it is not at all clear why control is often not seen as safety-critical.

3.9. How Does Size Correlate with Complexity?

There is still a perception that size correlates with complexity. How do we deal with this?

- It's really a matter of semantics versus structure.
- The real problem is the functional complexity and we don't have a handle on that at all.
- There is no correlation between complexity metrics and errors.
- On the other hand, there are likely to be more errors in large systems than in small ones.
- In a crude sense, 100,000 lines of code (LOC) is more complex than 100. However, just because one system has more LOC than another does not imply that its complexity is higher.
- If you increase the size, you do increase the number of problems that can occur; but small programs can also be very complex. Small is not necessarily better.
- It comes down to an intellectual assessment.

There was a follow-on discussion of fault tolerance and how it adds to complexity.

- Adding fault tolerance may add errors.
- Maybe the proper question is "can you provide an intellectual argument that this is correct?"
- There may be some merit to using complexity measures in a qualitative fashion but don't set things up so that there is an incentive to minimize size.
- Numbers on error density are averages and are not necessarily meaningful when applied to a particular program – there is wide variation.
- The application is the major influence on how many errors you have and what kinds of errors.
- The programmer has the biggest effect. There is a factor greater than 25 between the ability of the best and worst programmer.
- This is not convincing. In the systems we are discussing here, there will be no such thing as *the* programmer. In design teams of any size, such differences should average out.
- You probably can't certify programmers, but someone in charge on the project should be reviewed and have a certain level of experience.

3.10. Partitioning the Design

I think I heard someone say that it's 0 or 1; if the probability of being 0 is low, it's 1. What about partitioning the design?

- Can you really achieve a design like that?
- The Germans have thought about this.

- This is really a discussion on the point that there is a region where you understand the program completely (0), then you enter the realm of probability, and ultimately, there comes a point where you can't say anything at all about the program in a quantitative sense (1).

3.11. FMEA and Fault Tree Analysis

Are FMEA and fault tree analysis adequate?

- One panel member stated that he had never seen FMEA applied to software and doesn't know whether it can be done. Doing a FMEA or FMECA on software may not be realistic because of the extremely large number of possible software outputs and behaviors that would usually have to be considered.
- Verilog claims to have a tool to do this.

3.12. Unintended Functions

What defenses should be considered against unintended functions? How should they be addressed?

- If you can identify the unintended functions that you don't want, there are techniques to address this problem; for example, software fault tree analysis. It is not clear whether there are general techniques. Formal methods allow you to prove certain properties.
- You could use a safety kernel approach. If you build the kernel so that certain things can't happen, then it doesn't matter what the other components do.
- For Darlington, the AECB went through a lengthy process to sign off that there were no unintended functions.
- Showing that there are no unintended functions is very difficult – it's essentially a two-way proof between the code and the requirements.
- The notion of security kernel does not translate to safety. In a secure kernel you want to make sure that certain things never happen. Contrast this to a safety system where you want to be sure that certain things will happen.
- Security kernel ideas can be applied to safety. Safety usually involves ensuring that certain things can never happen; that is, that hazards don’t occur.

3.13. Convincing a Regulator That a Design is Correct

What would you consider to be the requirements for a design, development, and test process to convince a regulator that it is correct?

- High-level design, detailed design, and code with proofs that each step conforms with the previous step. This includes formal requirements specification, design specification, proof that the two specifications are equivalent, and proof that the design is equivalent to the implementation. You should have a detailed formal argument (full mechanical proof) and journal-style summary.
- You need to have both a proof with extensive detail and some way of understanding the proof at a higher level of abstraction. The proof needs to be presented hierarchically and from different points of view. You should avoid box loads of information. The proof has to be hierarchical. You need data for the entire system. The vendor must honor the intent and the spirit of the regulation, not just the letter of the regulation.
- Even with complete mechanical verification, it should be written up as journal-style proofs for ordinary readers. The written proofs summarize things for people to understand.
- Have you ever seen anyone do that?
- For an entire system, no. For critical subsystems, yes.
- Yes – and I've found errors in the journal-style arguments!
- You need to encourage the vendor to honor the intent rather than the letter of the regulation. You really need to understand why this thing works – all the rest is really there to support this.

3.14. Testing

What about test? Should the regulator require testing?

- The panel didn't mean to imply that you shouldn't test, just that there wasn't anything new to add here.

At this point, there was an elaboration on comments on criticality levels.

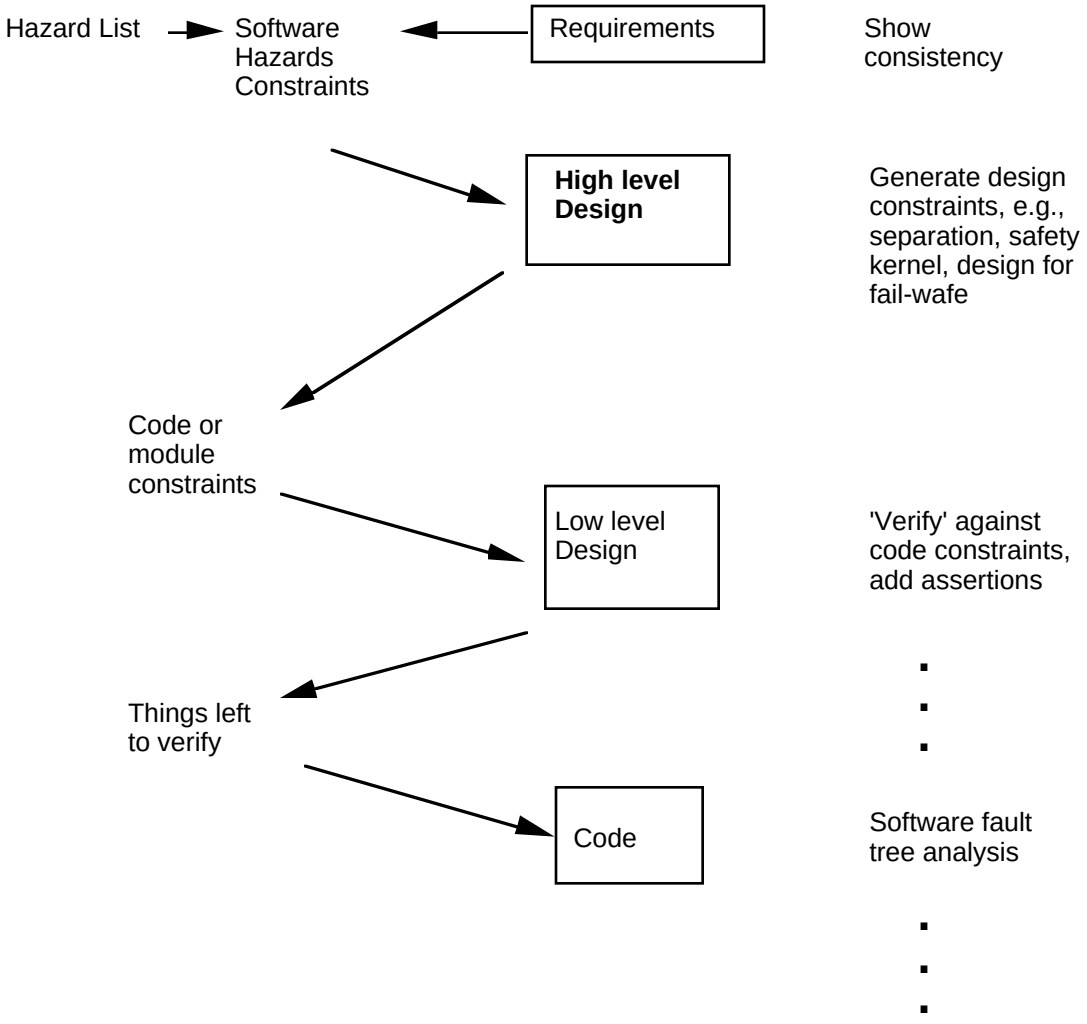
- If a system component is of the highest criticality level, then software that contributes to a failure of that system must also be considered to be at that level.
- The usual approach is to develop a hazard list, match it to components and then require the best techniques for the most critical components. This leads to redefining things in terms of reliability and forgetting the hazards. Since ultra-high reliability is impossible to achieve or guarantee for software, this paints you into a corner where safety depends on something that is impossible to do and your only option is to produce perfect software. Most software engineering techniques have never been shown to be effective enough to achieve perfection.
- The usual method of verification is concerned with showing that the code is consistent with the specifications. If you wish to include unintended functions, you also need to do the reverse.
- Another way is to show consistency between requirements and safety constraints. Do the usual verification between specifications, design, and code; then verify the code against safety constraints. This saves a lot of work.
- A more general principle is don't write standards that prohibit good solutions.
- Use a hazard list to identify software hazards, and generate the list of constraints from this.
- Show consistency between software requirements and the constraints generated from the list of hazards.
- Use software hazards to control the design of the software (for example, separation, encapsulation kernel, safety kernel, design for fail-safe, ...).
- From high-level design, generate and verify code or module constraints.
- "Verify" low-level design against code constraints; then add assertions.
- Starting with the mindset that the system has to be correct leads to one set of choices. If you start with a different mindset (safety), you get a different set of choices. It does not need to be correct to be safe.
- (from DO 178B) Start with a list of system hazards. Assign a criticality level to pieces of code based upon what they can affect. Various lists of techniques exist associated with these levels (for example formal specification, black box testing techniques, white box testing techniques, etc.). The highest level must be correct. After dealing with a hazard initially at the system level then map to the design and the code. Then forget about the hazards because you've assigned a criticality level and set of techniques to match. You don't need to build test cases on the basis of safety properties. Just work on reliability.
- There is also an opposite approach. Prove the requirements are consistent with the safety constraints. Then use the safety constraints to guide the design of the software so that the software, even though it may contain errors, will not contribute to an accident. The software is, in essence, fail-safe.
- Fault trees work backward from the hazard and are better than FMEA.
- If software must take safety action, then you have to show that it will always perform when called upon to do so. The Darlington system design was smart in that it started "tripped", for example in a safe state. So they needed to show that no subroutine would untrip one of these variables when it wasn't supposed to. They built a fault tree to see that it would not trip when it wasn’t supposed to. They found no such problem but did find 42 changes to make that increased the safety of the code.
- The illustration on the next page shows one way to view the interactions among hazards, constraints and software design elements.

3.15. Using a Safety Approach to Improve the Reliability Numbers

If I do everything right and use this safety approach, do I get from 10^{-4} to 10^{-5}?

- Those are different numbers; one is reliability and one is safety. It might increase your confidence but it can't give you a number.
- Fault trees are a logical argument, and reliability figures don't address this question.

- You can't assign a reliability figure to the software events that populate the fault tree. (There was a great deal of disagreement on this point.)
- Combining these kinds of information is tricky – for example, the testing may give you 10^{-4} and the safety analysis may tell you *why* it's 10^{-4}. One panel member thought this was "combining apples and oranges" and the result did not provide the information needed.
- All this safety analysis work just tells you why you got to 10^{-4} in the first place. The point is that you must rely on a combination of evidence that tells you something new, but we still don't know how best to combine the different types of evidence or how to tell or measure what such a combination will get us. You can't quantify the end result.
- If you do reliability analysis and then safety analysis and find no new problems, it increases your confidence.
- If you find problems, it may also increase your confidence since they were different problems found by different techniques.
- The issue in both these cases is whether the increased confidence is scientifically justified. This is doubtful. It needs to be emphasized that there is evidence that people tend to have their confidence increased in circumstances where this is not really justified. We must distinguish between expert judgment and "factually, scientifically, informed" expert judgment.



Nancy Leveson's Development Approach

- Using fault trees helps find errors because it forces you to look at the code in a new way. Usually you focus on what you want the software to do; with a fault tree, you focus on what you *don't* want it to do.

- One approach is to use multiple but complementary techniques. This can increase your confidence, but you can't quantify the result.

3.16. Final Remarks

How about some last words of wisdom?

- The panel began by asking the observers: what did you all get from this? They then gave their own assessments of the workshop. The first set of remarks (preceded by '-') are from the observers. The second set (preceded by '•') are from the panel members.

  - A lot of the techniques are really good practice; there is no big winner. This creates a problem for the NRC since there are proposals on the table that must be answered, and there are time constraints on answering them.
  - A key point is agreeing to common standards and negotiating the implementation. Can we win the political battles that this will create? How does the NRC get the authority it needs?
  - We need to augment our existing approach with new tools. There was a consensus that it is reasonable for the NRC to ask for and require vendors to provide sufficient information to convince the NRC staff that they have addressed the common mode failures. Levels of diversity are also reasonable. Use manual backups. There will be some debate on degree, but this is a good path to follow. Also, looking at other tools and techniques, for example formal methods, is certainly very reasonable.
  - There was a good discussion on hazards analysis, and a good emphasis on intellectual arguments even though it's hard to derive regulatory policy. We need to do more thinking on levels of defense. I like the idea of "don't prohibit other solutions."
  - We have to do a "competency assessment" of vendors. The real value is: "do you understand it?" What is a good competency audit? The group's position with respect to the intellectual content of the software and the process of building it was interesting.

- It's interesting that the people who build reactor systems are not visible in the safety, reliability and formal methods communities. They are not involved in the intellectual debate. The ability and the exercise of making a formal argument or some kind of rigorous argument is essential. It's a matter of the vendor delivering not just the product but also being able to say "this is what we did" and "this is why we did it this way. "
- The NRC should start any assessment with extreme skepticism and put the burden of proof onto the vendor.
- The avionics industry may have gone too far in that the people who write the software also help write the standards.
- How practical is it to require formal methods, and will vendors actually use them? How will the NRC know whether they've been used competently? There is an essential need to do things quantitatively. However there are inherent limits in what we can do quantitatively. So we will have to make decisions based on disparate evidence. (This remark includes statistics and probability in formal methods.)
- Measurements are needed. A point of disagreement is whether we have them now. Be realistic about using numbers just because you want numbers when you need to factor in qualitative data. Don't do things in software that are simpler to do in hardware. Be sure that who you hire is qualified and not a "garage hacker."
- In other engineering fields, you require evidence rather than specific techniques; it depends on what's reasonable. You need someone to provide the argument and someone to scrutinize it.
- The NRC should be skeptical about replacing hardware with software, since it's not always advantageous. Not all formal methods are equal. You will need to do an in-depth investigation of formal methods to get a handle on this.
- The NRC will need to do things quantitatively, but there are inherent limits that must be faced.
- We need measurements. Since you need to make decisions now, qualitative judgments are better than bad numbers. If someone can't make a rational argument for correctness, they shouldn't be building the software.

4. Thursday Afternoon NRC / LLNL Discussion

During a wrap-up session Thursday afternoon, the NRC and LLNL observers discussed the results of the workshop among themselves. This portion of the report gives a summary of that discussion. Most of this discussion consisted of questions that will need to be examined (based on the discussions in the workshop) and actions that will need to be taken. Comments from panelists who read a draft of this Report are included in a few places.

4.1. General Points

- We need to revisit defense-in-depth. We need to look at the whole picture. As we move from analog to digital systems, we may need to re-examine the whole system and redo the hazards analysis.
- What do we think about hazards? Should they be in the forefront? Do we believe what the panel is telling us? Should we add in hazards analysis to the NUREG 0493 analysis, based on what was learned at this workshop? What is being done today on retrofits with respect to hazards?

4.2. Assessing Organizations

- Putting the vendor through more intellectual effort will increase our confidence in his system. How does one do a competency assessment? Can one put together a protocol for competency assessment?
- The NRC needs to figure out what it needs in a development organization before trying to measure the organization's competency.
- One possibility is to use the Software Engineering Institute (SEI) maturity model as a model for assessing organizations. Don't try to use it directly (since it was developed for a different objective - cost containment), but as an approach. Could we create a checklist of things to look for in organizations?

  A comment was added by a panel member during a review of a draft of this report: "I am very skeptical about this. It seems to me just yet more software engineering snake-oil. As far as I can judge, it [the SEI model] is only influential because the DoD forces it on suppliers. There seems to be no real empirical underpinning for the model." Another panelist said: "Arg! The SEI maturity model has never been shown to be effective. At best it might decrease cost, but will probably have little effect on quality."
- If we do this, we will need to give a vendor fair warning of what the NRC will expect to see in an assessment.
- It is clear that the intellectual capability of the vendor is of key importance.
- We have heard from the experts that it is necessary to be able to assess the intellectual products of the development effort. We have concluded that in order to do this, we need to assess the development process. We are aware that various assessment methods exist - HP, SEI, JPL and others. Can these be adapted to our needs? We need to decide how we are to do the assessment.

  Panel member: "I think it is fair to say that not a lot of usable empirical evidence has come from these studies. But a proper evaluation of the efficacy of different methods and processes needs to be done."

4.3. Methodology

- We kept hearing three things regarding reliability measurement: intellectual effort, formal methods and testing.
- The experts were assuming we understood the need to do the standard things such as life cycle processes, testing, and so forth. They were talking about what is needed *in addition* for safety critical systems.

  Panel member: "Yes, this is an important point - I think that ran through all the discussions."
- Remember the comments that the real benefit from using formal methods is the impact it has on your thinking process.

- Based on the results of this workshop, formal methods is an acceptable method for developing safe software. The reason is that the intellectual exercise increases knowledge of the software. The panel members were uniform in recommending formal methods. The problem is: What does this mean? The full approach (with complete proofs of correctness) is probably too much. Two questions arise: (1) How do you actually apply formal methods? (2) Are other methods equally applicable?
- The panel members emphasized that safety is a system issue. Diversity is essential because you can't get better than 10^{-4} reliability on individual software components.
- You can't test beyond 10^{-4}. If your design requires more than this, reject the design. You need to start with the hazard and figure out what probability of system failure is acceptable for the system. Then look at the system design and decide failure rates for the different components. If one of these components is software, and the required failure rate is lower than 10^{-4} (say, 10^{-5}), you can't do it in practice. Redesign. If the required failure rate is higher than 10^{-4} (say, 10^{-3}), then this can be tested, so the design is OK (from this standpoint).
- All the experts agreed that hazard analysis is needed. Where should it be done? Early in the system design, since it is a system issue. We need to address the identification of system hazards and the use of system hazards to identify software hazards. The concept of "software systems safety" rather than just "software safety" is an important one.
- Hazard analysis must be done both top-down and bottom-up. The former addresses system hazards that are dealt with by the software. The latter ensure that software faults don't impact system safety.
- Functional complexity is more important that structural complexity. The former is harder to measure; the latter is easier to measure. (This does not imply that measuring structural complexity is necessarily easy.)
- It was disappointing that there was no discussion of requirements validation. It is clearly an important area. There are techniques (such as simulation and rapid prototyping) that address this. We need to explore this further.
- The difference in viewpoint between a safety or hazards approach and a reliability approach is an important one.
- When assessing reliability, you are looking for what the system will do. When assessing safety, you are looking for what the system will not do.
- Panel member: "One particular bête-noir of mine got mentioned only briefly - that is the dearth of real hard evidence from operational use of these safety critical systems. I think it is scandalous that there are not more mandatory reporting requirements so that the community can learn about what is really happening to systems in the field. For example, I think someone said that there is no requirement to report when the protection system has been called upon to act, so long as it functions OK. This is crazy! We badly need complete reporting of all incidents."

[End]